AI Assistant Coding

Assignment 8

Name : GOLI NITHYA HT. No : 2303A52066 **Batch:** 32

Task Description #1 (Username Validator – Apply AI in Authentication Context)

• **Task:** Use AI to generate at least 3 assert test cases for a function is_valid_username(username) and then implement the function using Test-Driven Development principles.

• **Requirements:**

o Username length must be between 5 and 15 characters.

o Must contain only alphabets and digits.

o Must not start with a digit.

o No spaces allowed.

Example Assert Test Cases:

assert is_valid_username("User123") == True

assert is_valid_username("12User") == False

assert is_valid_username("Us er") == False

Expected Output #1:

• Username validation logic successfully passing all AI-generated test cases.

Code:

```
def is_valid_username(username):
    # Check if length is between 5 and 15 characters
    if len(username) < 5 or len(username) > 15:
        return False

    # Check if contains only alphabets and digits
    if not username.isalnum():
        return False

    # Check if starts with a digit
```

```
    if username[0].isdigit():
        return False

    # Check for spaces (implicit in isalnum() but being explicit)
    if ' ' in username:
        return False

    return True


# Test cases
assert is_valid_username("user123") == True
assert is_valid_username("123user") == False
assert is_valid_username("user name") == False
assert is_valid_username("user_name") == False
assert is_valid_username("user") == False
```

Task Description #2 (Even–Odd & Type Classification – Apply AI for Robust Input Handling)

• **Task:** Use AI to generate at least 3 assert test cases for a function classify_value(x) and implement it using conditional logic and loops.

• **Requirements:**

o If input is an integer, classify as "Even" or "Odd".

o If input is 0, return "Zero".

o If input is non-numeric, return "Invalid Input".

Example Assert Test Cases:

assert classify_value(8) == "Even"

assert classify_value(7) == "Odd"

assert classify_value("abc") == "Invalid Input"

Expected Output #2:

• Function correctly classifying values and passing all test cases.

Code:

```
def classify_value(value):
```

```
    """
    Classifies the input value into one of the following categories:
    - "Positive" if the value is greater than 0
    - "Negative" if the value is less than 0
    - "Zero" if the value is exactly 0
    """
    if value == 0:
        return "Zero"
    elif not isinstance(value, (int, float)):
        return "Invalid Input"
    elif value%2==0:
        return "Even"
    else:       return "Odd"
# Test cases
assert classify_value(10) == "Even"
assert classify_value(-5) == "Odd"
assert classify_value(0) == "Zero"
assert classify_value(3.5) == "Odd"
assert classify_value("string") == "Invalid Input"
```

Task Description #3 (Palindrome Checker – Apply AI for String Normalization)

• **Task:** Use AI to generate at least 3 assert test cases for a function is_palindrome(text) and implement the function.

• **Requirements:**

o Ignore case, spaces, and punctuation.

o Handle edge cases such as empty strings and single characters.

Example Assert Test Cases:

assert is_palindrome("Madam") == True

assert is_palindrome("A man a plan a canal Panama") ==True

assert is_palindrome("Python") == False

Expected Output #3:

• Function correctly identifying palindromes and passing all AI-generated tests.

Code:

```
# Ignore case, spaces, and punctuation.
# o Handle edge cases such as empty strings and single
# characters.
import string

def check_palindrome(s):
    if not isinstance(s, str):
        raise ValueError("Input must be a string")
    if s == "" or len(s) == 1:
        return True
    # Remove punctuation
    s = s.translate(str.maketrans('', '', string.punctuation))
    # Remove spaces and convert to lowercase
    cleaned_s = s.replace(" ", "").lower()

    # Check if the cleaned string is equal to its reverse
    return cleaned_s == cleaned_s[::-1]

# Test cases
assert check_palindrome("A man a pla'n a canal Panama") == True
assert check_palindrome("Hello World") == False
assert check_palindrome("Madam") == True
assert check_palindrome("python") == False
```

Task Description #4 (Email ID Validation – Apply AI for Data Validation)

• Task: Use AI to generate at least 3 assert test cases for a function validate_email(email) and implement the function.

• **Requirements:**

o Must contain @ and .

o Must not start or end with special characters.

o Should handle invalid formats gracefully.

Example Assert Test Cases:

assert validate_email("user@example.com") == True

assert validate_email("userexample.com") == False

assert validate_email("@gmail.com") == False

Expected Output #5:

• Email validation function passing all AI-generated test cases and handling edge cases correctly.

Code:

```
# Must contain @ and .
# o Must not start or end with special characters.
# o Should handle invalid formats gracefully.

def is_valid_email(email):
    # Check if email contains exactly one '@' symbol
    if email.count('@') != 1:
        return False
    # Split the email into local part and domain part
    local_part, domain_part = email.split('@')
    # Check if local part and domain part are not empty
    if not local_part or not domain_part:
        return False
    # Check if domain part contains at least one '.' symbol
    if '.' not in domain_part:
        return False
    if domain_part.startswith('.') or domain_part.endswith('.'):
        return False
    # Check if email starts or ends with special characters
    if email[0] in ['@', '.'] or email[-1] in ['@', '.']:
        return False
    return True

# Test cases
assert is_valid_email("user@example.com") == True
assert is_valid_email("userexample.com") == False
assert is_valid_email("user@.com") == False
assert is_valid_email("@example.com") == False
```

Task 5 (Perfect Number Checker – Test Case Design)

• Function: Check if a number is a perfect number (sum of divisors = number).

• **Test Cases to Design:**

o Normal case: 6 → True, 10 → False.

o Edge case: 1.

o Negative number case.

o Larger case: 28.

• **Requirement:** Validate correctness with assertions.

Code:

```
def is_perfect_number(n):
    if n < 2:
        return False
    divisors_sum = 1  # Start with 1, which is a proper divisor
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            divisors_sum += i
            if i != n // i:  # Add the complementary divisor if it's different
                divisors_sum += n // i
    return divisors_sum == n

# Test cases
# Normal cases - Positive perfect numbers
assert is_perfect_number(6) == True, "6 should be a perfect number (1+2+3=6)"
assert is_perfect_number(28) == True, "28 should be a perfect number (1+2+4+7+14=28)"
assert is_perfect_number(496) == True, "496 should be a perfect number"

# Normal cases - Non-perfect numbers
assert is_perfect_number(10) == False, "10 should not be a perfect number (1+2+5=8≠10)"
assert is_perfect_number(12) == False, "12 should not be a perfect number (1+2+3+4+6=16≠12)"

# Edge case: 1
assert is_perfect_number(1) == False, "1 should not be a perfect number (no proper divisors)"

# Edge case: 0
assert is_perfect_number(0) == False, "0 should not be a perfect number"

# Negative number cases
assert is_perfect_number(-6) == False, "Negative numbers cannot be perfect numbers"
assert is_perfect_number(-28) == False, "Negative numbers cannot be perfect numbers"
assert is_perfect_number(-1) == False, "Negative numbers cannot be perfect numbers"

print("✓ All test cases passed successfully!")
```

Task 6 (Abundant Number Checker – Test Case Design)

• Function: Check if a number is abundant (sum of divisors >number).

• **Test Cases to Design:**

o Normal case: 12 → True, 15 → False.

o Edge case: 1.

o Negative number case.

o Large case: 945.

Requirement: Validate correctness with unittest

Code:

```
import unittest

def is_abundant_number(n):
    if n < 12:
        return False
    divisors_sum = 1  # Start with 1, which is a proper divisor
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            divisors_sum += i
            if i != n // i:  # Add the complementary divisor if it's different
                divisors_sum += n // i
    return divisors_sum > n

# Test cases Unit test class
class TestAbundantNumber(unittest.TestCase):
    def test_abundant_numbers(self):
        self.assertTrue(is_abundant_number(12), "12 should be an abundant number (1+2+3+4+6=16>12)")
        self.assertTrue(is_abundant_number(18), "18 should be an abundant number (1+2+3+6+9=21>18)")
        self.assertTrue(is_abundant_number(20), "20 should be an abundant number (1+2+4+5+10=22>20)")

    def test_non_abundant_numbers(self):
        self.assertFalse(is_abundant_number(6), "6 should not be an abundant number (1+2+3=6=6)")
        self.assertFalse(is_abundant_number(28), "28 should not be an abundant number
(1+2+4+7+14=28=28)")
        self.assertFalse(is_abundant_number(496), "496 should not be an abundant number")

    def test_edge_cases(self):
        self.assertFalse(is_abundant_number(1), "1 should not be an abundant number (no proper divisors)")
        self.assertFalse(is_abundant_number(0), "0 should not be an abundant number")
        self.assertFalse(is_abundant_number(-12), "Negative numbers cannot be abundant numbers")
        self.assertFalse(is_abundant_number(-18), "Negative numbers cannot be abundant numbers")
        self.assertFalse(is_abundant_number(-1), "Negative numbers cannot be abundant numbers")

if __name__ == '__main__':
    unittest.main()
```

Output:

```
(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe
d:/Course/AIAC/aiac-18-2-2026/abudent_number.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Task 7 (Deficient Number Checker – Test Case Design)

• Function: Check if a number is deficient (sum of divisors < number).

• **Test Cases to Design:**

o Normal case: 8 → True, 12 → False.

o Edge case: 1.

o Negative number case.

o Large case: 546.

Requirement: Validate correctness with pytest.

Code:

```
def is_deficient(n):
    if n < 1:
        return False
    divisors_sum = sum(i for i in range(1, n) if n % i == 0)
    return divisors_sum < n
# Test cases
def test_Normal():
    assert is_deficient(8) == True
    assert is_deficient(12) == False
    assert is_deficient(15) == True
    assert is_deficient(28) == False
    assert is_deficient(1) == True
def test_Edge():
    assert is_deficient(0) == False
    assert is_deficient(-5) == False
    assert is_deficient(2) == True
def test_negative():
    assert is_deficient(-1) == False
    assert is_deficient(-10) == False
def test_large():
```

```
    assert is_deficient(100) == False
    assert is_deficient(1000) == False
```

Output:

```
(AIAC_env) PS D:\Course\AIAC\aiac-18-2-2026> python -m pytest deficient_number.py
========================= test session starts =========================
platform win32 -- Python 3.14.0, pytest-9.0.2, pluggy-1.6.0
rootdir: D:\Course\AIAC\aiac-18-2-2026
collected 4 items

deficient_number.py ....                                         [100%]

========================== 4 passed in 0.04s ==========================
(AIAC_env) PS D:\Course\AIAC\aiac-18-2-2026>
```

Task 8 :

Write a function LeapYearChecker and validate its implementation using 10 pytest test cases

Code:

```
def LeapYearChecker(year):
    """
    Determines if a given year is a leap year.

    A leap year is defined as:
    - It is divisible by 4;
    - However, if it is divisible by 100, it must also be divisible by 400 to be a leap year.

    Parameters:
    year (int): The year to check.

    Returns:
    bool: True if the year is a leap year, False otherwise.
    """
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return True
    return False

def test_typical_leap_year_2020():
    """Test case 1: Year 2020 - divisible by 4, not by 100"""
    assert LeapYearChecker(2020) == True
```

```
def test_typical_leap_year_2024():
    """Test case 2: Year 2024 - divisible by 4, not by 100"""
    assert LeapYearChecker(2024) == True


def test_century_leap_year_2000():
    """Test case 3: Year 2000 - divisible by 400"""
    assert LeapYearChecker(2000) == True


def test_century_leap_year_1600():
    """Test case 4: Year 1600 - divisible by 400"""
    assert LeapYearChecker(1600) == True


def test_non_leap_year_2019():
    """Test case 5: Year 2019 - not divisible by 4"""
    assert LeapYearChecker(2019) == False


def test_non_leap_year_2021():
    """Test case 6: Year 2021 - not divisible by 4"""
    assert LeapYearChecker(2021) == False


def test_century_non_leap_year_1900():
    """Test case 7: Year 1900 - divisible by 100 but not by 400"""
    assert LeapYearChecker(1900) == False


def test_century_non_leap_year_2100():
    """Test case 8: Year 2100 - divisible by 100 but not by 400"""
    assert LeapYearChecker(2100) == False


def test_edge_case_year_1():
    """Test case 9: Year 1 - very early year, not divisible by 4"""
    assert LeapYearChecker(1) == False


def test_edge_case_year_4():
    """Test case 10: Year 4 - smallest positive leap year"""
    assert LeapYearChecker(4) == True
```

Output:

```
(AIAC_env) PS D:\Course\AIAC\aiac-18-2-2026> python -m pytest leapCheck.py
========================= test session starts =========================
platform win32 -- Python 3.14.0, pytest-9.0.2, pluggy-1.6.0
```

```
rootdir: D:\Course\AIAC\aiac-18-2-2026
collected 10 items

leapCheck.py ..........                                  [100%]

====================== 10 passed in 0.03s ======================
```

Task 9 :

Write a function SumOfDigits and validate its implementation using 7 pytest test cases.

Code:

```
def sumOfDigits(n):
    return sum(int(digit) for digit in str(n))

#test cases

def test_case1():
    """Test case 1: sumOfDigits(123) should return 6 (1 + 2 + 3)"""
    assert sumOfDigits(123) == 6
def test_case2():
    """Test case 2: sumOfDigits(0) should return 0"""
    assert sumOfDigits(0) == 0
def test_case3():
    """Test case 3: sumOfDigits(999) should return 27 (9 + 9 + 9)"""
    assert sumOfDigits(999) == 27
def test_case4():
    """Test case 4: sumOfDigits(12345) should return 15 (1 + 2 + 3 + 4 + 5)"""
    assert sumOfDigits(12345) == 15
def test_case5():
    """Test case 5: sumOfDigits(1001) should return 2 (1 + 0 + 0 + 1)"""
    assert sumOfDigits(1001) == 2
def test_case6():
    """Test case 6: sumOfDigits(987654321) should return 45 (9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1)"""
    assert sumOfDigits(987654321) == 45
def test_case7():
    """Test case 7: sumOfDigits(111) should return 3 (1 + 1 + 1)"""
    assert sumOfDigits(111) == 3
```

Output:

```
(AIAC_env) PS D:\Course\AIAC\aiac-18-2-2026> python -m pytest sumOfDigits.py
```

```
========================================= test session starts
=========================================
platform win32 -- Python 3.14.0, pytest-9.0.2, pluggy-1.6.0
rootdir: D:\Course\AIAC\aiac-18-2-2026
collected 7 items

sumOfDigits.py .......                                          [100%]

========================================= 7 passed in 0.02s
=========================================
(AIAC_env) PS D:\Course\AIAC\aiac-18-2-2026>
```

Task 10 :

Write a function SortNumbers (implement bubble sort) and validate its implementation using 25 pytest test cases.

Code:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
# Test cases 25
def test_case1():
    arr = [64, 34, 25, 12, 22, 11, 90]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [11, 12, 22, 25, 34, 64, 90]
def test_case2():
    arr = [5, 1, 4, 2, 8]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 4, 5, 8]
def test_case3():
    arr = [3, 0, 2, 5, -1, 4, 1]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [-1, 0, 1, 2, 3, 4, 5]
def test_case4():
    arr = [1, 2, 3, 4, 5]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 3, 4, 5]
def test_case5():
    arr = [5, 4, 3, 2, 1]
    sorted_arr = bubble_sort(arr)
```

```
    assert sorted_arr == [1, 2, 3, 4, 5]
def test_case6():
    arr = [1]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1]

def test_case7():
    arr = []
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == []
def test_case8():
    arr = [2, 3, 2, 1, 4]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 2, 3, 4]
def test_case9():
    arr = [5, 1, 4, 2, 8, 5]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 4, 5, 5, 8]
def test_case10():
    arr = [3, 0, 2, 5, -1, 4, 1, 3]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [-1, 0, 1, 2, 3, 3, 4, 5]
def test_case11():
    arr = [1, 2, 3, 4, 5, 6]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 3, 4, 5, 6]
def test_case12():
    arr = [6, 5, 4, 3, 2, 1]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 3, 4, 5, 6]
def test_case13():
    arr = [1, 1, 1, 1, 1]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 1, 1, 1, 1]
def test_case14():
    arr = [5, 4, 3, 2, 1, 0]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [0, 1, 2, 3, 4, 5]
def test_case15():
    arr = [0, 1, 2, 3, 4, 5]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [0, 1, 2, 3, 4, 5]
def test_case16():
    arr = [3, 2, 1, 0, -1]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [-1, 0, 1, 2, 3]
def test_case17():
    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
def test_case18():
    arr = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
def test_case19():
    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
def test_case20():
    arr = [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
def test_case21():
    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
def test_case22():
    arr = [12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
def test_case23():
    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
def test_case24():
    arr = [13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
def test_case25():

    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
    sorted_arr = bubble_sort(arr)
    assert sorted_arr == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Output:

```
(AIAC_env) PS D:\Course\AIAC\aiac-18-2-2026> python -m pytest bubbleSort.py
========================================= test session starts
=========================================
platform win32 -- Python 3.14.0, pytest-9.0.2, pluggy-1.6.0
rootdir: D:\Course\AIAC\aiac-18-2-2026
collected 25 items

bubbleSort.py .........................                                  [100%]
```

```
========================================== 25 passed in 0.05s
==========================================
(AIAC_env) PS D:\Course\AIAC\aiac-18-2-2026>
```

Task 11 :

Write a function ReverseString and validate its implementation using 5 unittest test cases

Code:

```
import unittest
def reverse_string(s):
    return s[::-1]
# Example usage:
class TestReverseString(unittest.TestCase):
    def test_reverse_string(self):
        self.assertEqual(reverse_string("hello"), "olleh")
    def test_empty_string(self):
        self.assertEqual(reverse_string(""), "")
    def test_single_character(self):
        self.assertEqual(reverse_string("a"), "a")
    def test_palindrome(self):
        self.assertEqual(reverse_string("madam"), "madam")
    def test_with_spaces(self):
        self.assertEqual(reverse_string("hello world"), "dlrow olleh")
if __name__ == '__main__':
    unittest.main()
```

Output:

```
(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe
d:/Course/AIAC/aiac-18-2-2026/reverseString.py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

Task 12 :

Write a function AnagramChecker and validate its implementation using 10 unittest test cases.

Code:

```
import unittest

def is_anagram(s1, s2):
    # Remove spaces and convert to lowercase
    s1 = s1.replace(" ", "").lower()
    s2 = s2.replace(" ", "").lower()

    # Sort the characters of both strings and compare
    return sorted(s1) == sorted(s2)
class TestAnagram(unittest.TestCase):
    def test_anagram(self):
        self.assertTrue(is_anagram("listen", "silent"))
    def test_not_anagram(self):
        self.assertFalse(is_anagram("hello", "world"))
    def test_anagram_with_spaces(self):
        self.assertTrue(is_anagram("conversation", "voices rant on"))
    def test_anagram_with_different_cases(self):
        self.assertTrue(is_anagram("Dormitory", "Dirty Room"))
    def test_anagram_with_numbers(self):
        self.assertTrue(is_anagram("123", "321"))
    def test_not_anagram_with_numbers(self):
        self.assertFalse(is_anagram("123", "456"))
    def test_anagram_with_special_characters(self):
        self.assertTrue(is_anagram("A!B@C#", "C#B@A!"))
    def test_not_anagram_with_special_characters(self):
        self.assertFalse(is_anagram("A!B@C#", "D#E@F!"))
    def test_anagram_with_unicode_characters(self):
        self.assertTrue(is_anagram("résumé", "sérumé"))
    def test_not_anagram_with_unicode_characters(self):
        self.assertFalse(is_anagram("accent", "accenté"))
if __name__ == "__main__":    unittest.main()
```

Output:

```
(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe
d:/Course/AIAC/aiac-18-2-2026/anagram.py
.........
----------------------------------------------------------------------
Ran 10 tests in 0.001s

OK
```

Task 13 :

Write a function ArmstrongChecker and validate its implementation using 8 unittest test cases.

Code:

```
import unittest
def is_armstrong_number(num):
    if not isinstance(num, int) or num < 0:
        raise ValueError("Input must be a non-negative integer")
    # Convert the number to a string to easily iterate over digits
    num_str = str(num)
    # Get the number of digits
    num_digits = len(num_str)

    # Calculate the sum of the digits raised to the power of the number of digits
    armstrong_sum = sum(int(digit) ** num_digits for digit in num_str)

    # Check if the calculated sum is equal to the original number
    return armstrong_sum == num
class TestArmstrongNumber(unittest.TestCase):
    def test_armstrong_numbers(self):
        self.assertTrue(is_armstrong_number(153))  # 1^3 + 5^3 + 3^3 = 153
        self.assertTrue(is_armstrong_number(370))  # 3^3 + 7^3 + 0^3 = 370
        self.assertTrue(is_armstrong_number(371))  # 3^3 + 7^3 + 1^3 = 371
        self.assertTrue(is_armstrong_number(9474)) # 9^4 + 4^4 + 7^4 + 4^4 = 9474

    def test_non_armstrong_numbers(self):
        self.assertFalse(is_armstrong_number(123))   # Not an Armstrong number
        self.assertFalse(is_armstrong_number(10))    # Not an Armstrong number
    def test_edge_cases(self):
        self.assertTrue(is_armstrong_number(0))     # 0^1 = 0, considered an Armstrong number
        self.assertTrue(is_armstrong_number(1))     # 1^1 = 1, considered an Armstrong number
    def test_large_numbers(self):
        self.assertFalse(is_armstrong_number(9475)) # Not an Armstrong number
        self.assertFalse(is_armstrong_number(10000)) # Not an Armstrong number
    def test_negative_numbers(self):
         with self.assertRaises(ValueError):
            is_armstrong_number(-153)  # Negative numbers are not Armstrong numbers
    def test_non_integer_input(self):
        with self.assertRaises(ValueError):
            is_armstrong_number("abc")  # Non-integer input should raise an error
    def test_float_input(self):
        with self.assertRaises(ValueError):
            is_armstrong_number(153.0)  # Float input should raise an error
    def test_large_armstrong_number(self):
        self.assertTrue(is_armstrong_number(9926315)) # 9^7 + 9^7 + 2^7 + 6^7 + 3^7 + 1^7 + 5^7 =
9926315
if __name__ == '__main__':
```

```
    unittest.main()
```

Output:

```
(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe
d:/Course/AIAC/aiac-18-2-2026/armstrong.py
........
----------------------------------------------------------------------
Ran 8 tests in 0.001s

OK
```